Teaching Machines and Programed Instruction

ROBERT T. FILEP, *Editor*

Controlling Behavior Changers' Behavior

DAVID G. MARKLE

Remarkably little behavioral technology has been applied systematically to controlling the programing behavior of instructional programmers. The programing systems which were assembled in the late 1950's were based upon theoretically promising ideas, but the functional rules and models which were given to programmers had many unfortunate effects. The first section of this paper reviews some unfortunate controlling effects of two major early programing methodologies. The second section outlines an approach which has been designed to provide beneficial programmer control. The final section discusses behavior maintenance contingencies which can increase the probability of sound instructional engineering.

EARLY METHODOLOGIES

Skinnerian Programing

It is appropriate to begin with the Skinnerian "linear" programing movement for several reasons. It provided the starting point for the work to be discussed in later sections of this paper. Also, it is the movement which should have been most concerned with programmer control, since its formulators were operant conditioners who freely used words like 'control' and who drew parallels between operant conditioning and programed instruction (e.g., Skinner, 1964).

David G. Markle is senior associate, Behavioral Engineering Associates, Los Altos, California; consultant to the Social and Educational Research Program, American Institutes for Research; consultant to the Center for Programmed Learning for Business, University of Michigan.

Although there were humanist objections to words like 'control,' and moral objections to the idea that learning should be easy and error-free, it was not obvious that the early Skinnerian formulation would have unfortunate effects on progammer behavior. Behavioral objectives should help the programmer weed out nonfunctional material and make it possible to validate instruction. Small steps and low error rate should enable the programmer to produce success where there had been failure. Frequent overt responses by the student should keep the programmer oriented to student behavior and give him detailed feedback on it. Similarly, one would have good expectations of self-pacing, logical sequencing, and immediate confirmation.[1]

But early programs appear in retrospect to be parodies of the basic ideas. Premature commercialization played a dominant role in this and has caused the field difficulties today. Many commercial programs give the critic an easy target and give the neophyte programmer a dysfunctional model. However, the mistakes which were made in the early days of the movement are related as much to theoretical issues as to premature commercialization.

The programmer was told to specify objectives in behavioral terms. Objectives couched in terms of "understanding" or "appreciation" were forbidden on the grounds of poor interobserver reliability. But the effect this had on many programmers was not to sharpen their analytical skills as intended, but rather to increase the frequency of less troublesome, trivial objectives. "The salesman will understand objection handling" would all too often shift to "The salesman will list the five key points of objection handling." Thus a prescription which was intended to improve instruction failed in many cases to bring about the intended improvement, not through a failure of the prescription, but of its implementation.

The programmer who operated with objectives which were related only to superficial verbal aspects of the desired behavior change had little guidance in what to ask the student to do at each step. Since each response request represents a fine-grained

[1] The questionable equation of formal confirmation and reinforcement which abounds in the early literature will not be dealt with here (Hamilton, 1964b). It has had little to do with faults of early programs, except in the unfortunate cases in which Skinner was misinterpreted to be prescribing *repetition* when he stressed reinforcement.

objective, weakness in the general objectives revealed itself in individual frames like the following:

> This is a picture of the model 5200 *sales register*. It is your friend. In this course you will learn to operate the model 5200 s——— r———.
>
> and
>
> A *noun* is the name of a person, place, or thing. The words 'cat' and 'mat' are n———s.

Programmers were told that responses should be relevant. But what is a relevant response? Within the model of filled-in blanks, programmers turned to the often irrelevant criterion, "important term." Thus 'noun,' 'accrued interest,' and 'manager' were typical responses in programs on grammar, investment, and business management. The requirement for learning to be comparatively free of errors further contributed to the problem. The following fictitious example is closer to the truth than one might imagine.

Suppose a programmer of psychology wishes to have the student identify a reinforcer as such. He might write 'The click of the food mechanism is a conditioned ————.' The student might respond 'stimulus,' which is technically correct, but not what the programmer wanted. In order to increase the probability of the desired response, the programmer might change the blank '————' to 'r————.' Then to confound things, the student might respond 'reward,' which is technically incorrect. To preempt this response, the programmer might add 'technical term' or, to carry it to the extreme, change the blank to 'rein——er.' The equally frustrated student might answer with 'de' instead of 'forc'!

The unfortunate translation of the low error criterion to one of eliminating errors at all costs produced many progressions of this absurd type as programs passed through developmental tryouts. And many programmers learned from experience or example to short circuit the tryout process by producing frames of the 'rein———er' type in their first drafts, thus avoiding the punishment of wrong answers from tryout students.

The mistake can be described in two ways. In terms of the observable behavior of the student, we can say that the addition

of formal prompts like 'r,' 'rein———er,' and the like simply changes the correct response from 'reinforcer' to 'einforcer' to 'forc.' The programmer did not increase the probability of the desired response; he changed responses. This same conclusion can be drawn in terms of the total behavioral episode, which includes unobservable thinking. The student could say to himself, "It begins with 'r,' " or "What word in psychology begins with 'rein' and ends with 'er'?" Unqualified insistence on correct observed responses often led to neglect of unobserved mediating behavior. Observable behavior was stressed for its own sake, not as evidence of a more important underlying behavioral episode.

Behavioral measures in snake phobia experiments illustrate this point (Bandura, Blanchard, & Ritter, 1967). The subject is asked to approach a cage which contains a live snake, reach in, pick up the snake, and play with it. If the experimenter substituted a plastic snake for the live snake in order to improve the apparent success of his cure, he would be making a similar mistake.

The basic ideas of the Skinnerian model need not have led programmers to produce trivial programs, and they did not uniformly. But the frequency of these effects suggests that there were serious gaps between the basic theoretical goals and their implementation. These gaps center around the difficulty of teaching programmers to do behavioral analysis. Since this difficulty has been so pervasive, some general aspects of the problem will be discussed below, to lay a groundwork for later sections of this paper.

The S-Man Problem

Evaluators of instructional materials can be classified according to where they look first. Asked to evaluate a lecturer, one might comment on his mannerisms, the clarity of his speech, his humor or lack thereof, his organization, his content, or whether he makes his material functionally relevant to student behavior. These different kinds of evaluation can be arranged roughly on a continuum, ranging from concern with superficial aspects of the lecturer's behavior, through content-related aspects, and finally to considerations of his effects on student behavior.

Instructional design problems can also be approached from different points on the continuum. One can first ask what audiovisual devices are available, how long the course should be, how to organize the content, what students should do to learn the material, or what the students' final behavior should

be. These and many other points must be considered at one time or another. But since decisions about any one of them will influence others or even preempt making decisions about others, the starting point is critical.

The continuum along which these different considerations range can be split into two basic sections, according to whether the main focus of analysis is on the instructional stimuli (S) or on the responses (R) the student is to make.[2] For convenience, a person who looks first at stimulus dimensions can be called an "S-man," while the person who looks first at response dimensions can be called an "R-man."[3] Thus an audiovisual person is typically a radical S-man, the more content-oriented instructor is a conservative S-man, while a programmer or behavioral engineer is or ought to be an R-man, who first specifies his goals in terms of what he wants his students to be able to do, then works back to the resources available to him on the "S" side of the continuum.

The major difficulties met by the Skinnerian approach appear to be a function of the strong S-man bias we have had at least since Socrates (whose token use of responses was hamstrung by the assumption that the student already "knew"). The simplest S-man mistake is the widespread assumption that programed instruction has to *look* a certain way. A tradition of evaluating textbooks, lectures, etc., by looking at them has made it natural to classify and evaluate programed instruction visually, by such irrelevant aspects as frame area and shape and number of words per frame.

Early trainers of programmers wanted their students to write frames which requested "relevant" responses. Instead, students of programing often produced materials which resembled lectures more than successive behavioral steps. Lacking effective ways to teach behavioral analysis, they fell back on an S-man approach and gave students artificial restrictions on format and

[2] This is an oversimplification, since treating responses *in vacuo* is the very mistake discussed earlier, of not accounting for changes in the discriminative stimuli which change the behavioral episode. In the current discussion, take "response" to include the episode and any critical SD's.

[3] Talk of S-men and R-men comes from Georgie Geis's more detailed discussions of four points of view in psychology, S (stimulus), O (organism), R (response), S (reinforcer), and the effects of focusing on any one in designing instruction. The R category in this discussion is broader than in Geis's, partly to simplify the exposition, and partly because a somewhat different point is being made.

the number of words they could use in a frame. These restrictions did little to strengthen the response component of programs. They simply translated the intuitive, response-related concept of step size into a rigid stimulus dimension. The neophyte programmer's natural first concern was to match the appearance of the model, not to engineer behavior. This has been called stuffing sheep into wolf's clothing.[4]

The basic reason for stressing an R-man approach in this paper lies in its strategic power to provide the programmer with a nonintuitive decision-making tool. For example, the concept of small step can be redefined as the largest step the student can take *successfully* if an empirical R-man strategy is adopted. The R-man strategy emphasized here is independent of the reasons which are often offered in support of either response-oriented or stimulus-oriented instruction (Klaus, 1965). Those arguments concern the nature of the final product–not the development process. There is clear evidence that *observable* responses are not necessarily required for learning to occur, although they are required for its measurement (Bandura, 1965). In most instances the important part of a response is what takes place *before* the overt behavior. Adoption of the R-man strategies discussed in the second section does not automatically preclude developing a stimulus-oriented product.

Consider, for example, using an R-man approach to produce a motion picture film which is intended for normal, noninterrupted viewing. Criterion items which require observable responses can be used throughout the development process, but can be dropped from the final version. A related procedure for making instruction more efficient is to convert criterion questions which have very low error frequencies to statement form in the final product (Markle, 1965a, 1967).

Crowderian Programing

In contrast with Skinner's behavioristic "control" model, Norman Crowder put forth a programing model based on "communication" and "feedback" ideas (Crowder, 1960). Although this model differs from the Skinnerian in basic theorizing and most details of execution, there are interesting parallels between the two in terms of the unfortunate controlling effects they had upon programmers.

The basic ideas of the Crowderian model are not without ap-

[4] N. Hamilton, personal communication, January 1968.

peal, although they lack the potential power of behavioral analysis and empirical development. In particular, the idea that a program should adapt to individual differences in more ways than rate of progress through a fixed sequence distinguishes the Crowderian model from the early Skinnerian model.

The Crowderian programmer was told to write a page of text, then pose a multiple-choice question to test a detail of the "communication" intended by the text. He was required to write different feedback for each choice, either confirming the correctness of the answer or admonishing the student for his incorrect answer. The "correct answer" feedback page would contain the next sequential material. The incorrect answer feedback pages for an item would reinstruct the student and, if necessary, lead him through a supplementary sequence of instructional steps to remedy his deficiency.

The practical problems which face the Crowderian programmer are formidable. First, he must predict errors in the manner of a multiple-choice test item writer. Second, he must write feedback and remedial branches which make students less likely to make those errors in the future. Lacking empirical data on which to base his programing, the programmer is under the control of unfortunate variables. A typical pattern is to omit critical information from one of the "communication" paragraphs in order to predict error reliably. Some critics label this technique sadistic, but consider the problem facing the poor programmer. He must write at least one paragraph of remedial text for each wrong choice. If he writes four-choice questions, three feedback paragraphs are required for wrong answers. He may also need to write remedial sequences. Thus at least three-fourths of the differential feedback and "branched" instruction he produces is devoted to wrong answers. It is the rare programmer who will design low error materials when under the control of these contingencies. Who would write a program with the expectation that a large proportion of it would go unread?

Since the first draft of a Crowderian program is intended to provide appropriate feedback for errors, the errors themselves are acceptable. Programmers were thus never forced to deal with one of the most important programmer control contingencies provided by the Skinnerian model: student feedback to the programmer. The next section develops this aspect of program design.

The lack of an empirical base permits the omission of a kind of analysis which is essential if branching is to accomplish its aims. According to the basic model, students are presented with different feedback and/or instruction on the basis of a single multiple-choice response. No data are provided by the system to enable the programmer to evaluate the reliability of these critical multiple-choice items. The unreliability of a single multiple-choice item makes this a critical flaw. Interestingly, many students have discovered this flaw themselves, realizing that it led programmers to skip them over material they did not know. As a result, students often read all pages in the book, right and wrong answer branches alike, to be certain not to miss anything.

Just as difficulties with the *application* of behavioral objectives (low error rate, etc.), cannot be taken to be criticisms of the basic Skinnerian ideas, the unfortunate effects of asking the programmer to write in the Crowderian branching mode in no way invalidate the desired goal of instruction which is adaptive to individual differences. The difficulty lies in the superficial formal model which, like the misinterpretations of the Skinnerian ideas, has led to undesirable programmer behavior. Some possibilities for empirically based applications of branching will be suggested in the following section.

A DEVELOPING METHODOLOGY

Early History

In 1962 at the University of Michigan a mixed group of specialists in Skinnerian programing and business management[5] cooperated in designing a week-long course in programed instruction for industrial training personnel. The course has been given monthly since then, and follow-up work has been done with many of the participants. As a result, staff members have had the chance to observe the real-world effects of what is taught and have been in a position to modify the methodology and the design of the course gradually in the light of this feedback.

In the early years of the workshops, needs for changes in the methodology were signaled by undesirable programmer behavior of the types discussed in the preceding section. Participants could learn readily to make things which looked like frames by following the inadvertent S-man prescriptions of the early methodology. Having just learned these new skills, they were then

[5] Initially, D. M. Brethower, D. G. Markle, G. A. Rummler, and A. W. Schrader, III. Since then, many contributions have been made by new staff members and associates.

understandably resistant to staff attempts to counter the control exercised by the methodology. The eventual solution was, of course, to change what was taught.

At first, the changes were primarily ones of deletion—deletion of programs which dealt with the more troublesome aspects of the early methodology. Gradually, materials which stressed analysis, criterion item design, and empirical revision were introduced (e.g., Brethower *et al*, 1963, 1965; Markle, 1965b). The most recently introduced lessons treat programing as a sequential information-gathering, decision-making process. Strong emphasis is placed on the use of decision strategies and student performance data to answer design questions which previously would have been answered by fiat.

Current Practice

The basic procedure is one of working systematically backward from a final test instrument. Objectives may initially be specified in verbal statements, and analysis may involve flow charts, matrices, and so on. However, the proof of objectives specification lies in designing an exhaustive set of final performance measures which simulate task performance as closely as cost effectiveness estimates indicate is needed. Even very carefully worded statements of behavioral objectives have been found to exercise inadequate control over subsequent programing behavior, whereas the actual response requests which will be given to students are difficult to ignore.

Readers may rightly object that this is simply teaching for the test. However, traditional objections to teaching for the test rest on the inadequacy of the test. Teaching for "transfer" involves two tests. One functions as a normal criterion test, while the other is a measure of performance on different but related tasks. In formal transfer experiments, both tests are of course developed and administered as part of the experimental design (Ellis, 1965). In normal educational practice, however, a "transfer" measure, which usually involves real-world performance, is seldom made explicit. For example, the civics teacher is unlikely to have behavioral measures of "good adult citizenship" which can be used either to guide explicitly the design of instruction or to determine for whom and to what extent the more global goals of the instruction have been met.

The methodology under discussion rests on the assumption that "both" tests should be dealt with explicitly by prespecifying the range of all relevant testing parameters, including those of

"transfer" measures. According to this view, a failure of performance to transfer from the final criterion test to real-world performance is potentially attributable to faulty selection of the range of one or more testing parameters or of unrealistic expectations for "transfer."[6] Descriptions of different methods of exhaustively specifying and sampling from the universe of test items can be found in Hamilton (1964a, 1968) and Markle (1967).

Once the final criterion measures have been developed, the designer can choose among several strategies to determine what to do next, according to his uncertainty, the potential seriousness of a mistake, and other economic considerations. To minimize the risk of erring and to maximize his knowledge of the student population, he can adopt a radical empirical strategy and administer subsets of the criterion measure to trained and untrained members of the population to gather data on their instructional needs. This "ideal" strategy may not be the best strategy under certain constraints. A more moderate but more uncertain approach would be to pretest only items which are judged potentially to be in the entering repertoires of the students. Testing of either kind typically leads to changes in the objectives.

Next, the programmer develops an evaluation structure around which the program will be built. The structure is composed of criterion items sampled from the final set and/or newly generated subordinate criterion items. These are arranged to form a sequence of subtests for the program. According to strategy decisions similar to the above, this sequence can range from coarse-grained to fine-grained, and may or may not be subjected to extensive empirical trial.

The most rigorous empirical strategy at this point involves testing a coarse-grained performance structure, composed only of major criterion items, then using the data to estimate where subordinate response requests are needed. This process can be thought of either as a further identification of entering behavior or as a first step in determining "step size" empirically. Alternatively the programmer can fill in the majority of the questions needed in the structure on the basis of analysis and intuition. He then converts the

[6] There may be other causes, such as inappropriate maintenance contingencies in the environment.

performance structure into a program by adding instructional materials.

Again, he has a number of strategies available. He may treat the bare performance structure as a first draft program and test it on students with the expectation that many errors will be made. This is the extreme version of the "lean" programing approach. A more moderate version of lean programing is to omit potentially needed teaching material only when in doubt about whether students will need it. Although this leaves the door open for rationalization about "needed" content, its inherent risks can be balanced by improved efficiency when a great deal is known about the student population or when the subject matter is entirely foreign to them. Regardless of strategy, however, the programmer has the performance structure to guide each small addition. He is never in the position of designing instructional materials, then having an irrelevant variable like the continuity of his prose or the cutting of his film determine what questions to ask his students. For that matter, he never has to write a "teaching frame" per se, since the addition of teaching material to a performance structure automatically converts some response requests to "teaching frames."

The emphasis upon successively approximating the needed instruction rests on two basic principles. The first is the arbitrary or philosophical one that student performance data should be used as much as possible in designing instruction. The second is a methodological point which is concerned with minimizing the limitations inherent in using student performance data as an information source.

Student performance data can be divided into two classes: correct answers and incorrect answers. Correct answers appear to indicate that the program is working; incorrect answers appear to indicate that it is not. This simple analysis does not hold up when the question of the legitimacy or spuriousness of the answer is introduced. Answers can be correct for legitimate or spurious reasons; similarly, they can be incorrect for legitimate or spurious reasons.

The legitimately correct answer indicates that the program is working, but unfortunately the programmer cannot easily distinguish between legitimately correct answers (right for the right reasons) and spuriously correct answers

(right for the wrong reasons). Even if we discount that difficulty, the correct answer cannot be relied upon as the positive feedback it appears to be. Correct answers, whether legitimate or spurious, do not signal either excess redundancy or irrelevant content.

Negative feedback, in the form of wrong answers, is potentially more useful to the programmer. Even spuriously wrong answers indicate that some kind of change is necessary, if only to clarify the response requests themselves. While incorrect answers can be depended upon to indicate needed changes or additions, they do not signal excess redundancy or irrelevant content any better than do correct answers. They are most valuable to the programmer before he has added large amounts of instructional material. Thus the objective of the lean programing strategy is to maximize the chances that the programmer will get reliable feedback on what to add or change and to minimize the chances that he will start off with excess material which will not be identified by student tryout data.

The process-product confusion has arisen with lean programing as it did with the early methodologies. The word 'lean,' intended to describe the programmer's first drafts and the deliberate use of parsimony in the early developmental stages, has been misinterpreted to describe the final product. There are those who feel that lean programing will produce dry, difficult, cut-to-the-bone, high error programs, rather than the intended well-tailored programs. There is no reason why the rigorously empirical lean programmer might not on some occasions find that his final product turns out to be far more detailed and redundant *seeming* than he ever predicted would be needed, simply because his prediction was based upon a misevaluation of the students' entering behavior.

A striking effect of this broad-scale use of tryout data to guide programmer decision making has been to relocate many decision points in the developmental process. For example, media decisions are best delayed until after fine-grained decisions about needed instructional content are made. Finalization of objectives is delayed until after several rounds of developmental tryouts. Decisions about "linear" vs. "branching" need never be made at all on a large scale. If empirical strategies are followed,

the decision is not whether to write a branching program but whether to use branching to handle a specific problem revealed by tryout data.

Elements of the developing methodology which have not been dealt with here include criterion measure and lesson design strategies which involve simulation and examples rather than rules and sequencing plans which involve both logical and empirical analysis. They also include some incompletely formulated but promising notions like specifying behavioral objectives in academic areas by looking for subject-matter relevant behaviors which experts engage in with high frequency. If an expert finds a specific activity worthwhile, perhaps the student will, if it can be engineered properly. For example, the historian engages in a kind of detective work with on-the-spot reports of historical events more frequently than he memorizes dates, presumably because it is more rewarding. This has clear implications for ways in which to specify objectives for students of history.

PROGRAMMER BEHAVIOR MAINTENANCE

The preceding sections of this paper have sketched out how interpretations of two methodologies had unfortunate effects on programmer behavior, and have outlined an emerging discipline which avoids some of these effects. The discussion has focused on immediate effects of the methodologies rather than on long-term effects, although many of the undesired immediate effects persist in the programmer's repertoire after an initial training period.

Even without a clear formulation of the essentially negative character of feedback and the resulting need for a parsimonious or "lean" approach, feedback from student tryouts was a critical feature of the early Skinnerian methodology. Yet it was the most fragile from the standpoint of programmer behavior maintenance. The contingencies which operated against good empirical work still apply to the current methodology, and are worth re-examining.

As discussed in the first section, student errors, which should have been welcomed in developmental tryouts, often functioned as punishment of the programmer. This tended to increase the frequency of trivially preempted errors and to decrease the probability that further empirical trials would be engaged in. A surprising additional result of the low error criterion was adopted by some commercial program pub-

lishers. In order to achieve an arbitrary low percent error value, they would add easy items to a sequence in order to spread the existing errors over a large base.

The use of errors as a deliberately sought guide to decision making as discussed in the previous section, has minimized their punishing effects for programmers who fully adopt the new methodology; but there are other contingencies. In industrial settings, real or perceived time and cost pressures discourage careful empirical work unless strong moves are made to counter these pressures. It has not been customary to do a careful economic analysis of the potential benefits of achieving one set of objectives versus another set of objectives, compared with the relative costs of achieving them. Nor has it been customary to balance the increased developmental costs of achieving an objective more efficiently against the savings in manpower costs which can result. The introduction and refinement of economic analyses of objectives and of the different strategies which can be used to achieve them should provide contingencies which encourage rather than discourage empirical development.

At first sight, it seems more difficult to apply economic criteria to the development of materials for the schools than for industry—and thus more difficult to justify rigorous empirical development. However, there is no reason why the value of student time cannot be used in the suggested economic model, to say nothing of using the estimated long-term value of the instruction.

Because economic analyses are not likely to affect programmer control contingencies in the near future as much as might be desired, short-range considerations are in order. The S-man problem applies to the evaluators of programmers' output as well as to programmers. Accustomed to evaluating a product in terms of its appearance rather than in terms of the behavior change it produces, supervisors of programmers and purchasers of instructional systems will reinforce attention to formal and stylistic aspects unless efforts are made to emphasize more relevant but less obvious aspects of instructional engineering. It is a double contingency management problem. The programmer must clarify his own function so that he will in turn receive appropriate reinforcement.

* * * * * *

Programmer behavior maintenance problems have until now been primarily involved with the simple issue of whether or not an empirical approach is used. As more varied methods of specifying objectives are developed and the range of empirical methods becomes better delineated, decision processes for the appropriateness of different approaches will become fundamental parts of instructional technology. An important criterion for evaluating these developments is the degree to which they control and maintain desirable programmer behavior in the face of the wide range of countercontingencies.

REFERENCES

Bandura, A. Vicarious processes: A case of no-trial learning. In L. Berkowitz (Ed.), *Advances in experimental social psychology.* Vol. II. New York: Academic Press, 1965. Pp. 1-55.

Bandura, A., Blanchard, E. D., & Ritter, B. J. The relative efficacy of desensitization and modeling therapeutic approaches for inducing behavioral, affective and attitudinal changes. Unpublished manuscript, Stanford University, 1967.

Brethower, D. M., Markle, D. G., Rummler, G. A., Schrader, A. W., III, & Smith, D. E. P. *Programmed learning: A practicum, Test Version* 3. Ann Arbor, Mich.: Center for Programmed Learning for Business, 1963.

————. *Programed learning: A practicum.* Ann Arbor, Mich.: Ann Arbor Publishers, 1965.

Crowder, N. A. Automatic tutoring by intrinsic programing. In A. A. Lumsdaine & Glaser (Eds.), *Teaching machines and programmed learning.* Washington, D. C.: National Education Association, 1960. Pp. 286-304.

Ellis, H. C. *The transfer of learning.* New York: Macmillan, 1965.

Hamilton, N. R. Effects of logical versus random sequencing of items in an autoinstructional program under two conditions of covert response. *J. educ. Psychol.*, 1964, *55*, 258-266. (a)

————. The role of reinforcement in meaningful verbal learning: An explication involving self-instructional programs. Paper read at the American Institute for Research scientific meeting, Washington, D. C., March 1964. (b)

————. Differential response to instruction as a function of spatial and verbal aptitudes. Unpublished doctoral dissertation, Stanford University, 1968.

Klaus, D. J. An analysis of programing techniques. In R. Glaser (Ed.), *Teaching machines and programed learning, II.* Washington, D. C.: National Education Association, 1965. Pp. 118-161.

Markle, D. G. Empirical film development. *Nat. Soc. for Programmed Instruction J.*, 1965, *4* (6), 9-11. (a)

———. *Programmed instruction: The development process.* 16 mm color motion picture film developed under Contract OE 3-16-036, Title VII-B, Public Law 85-864 for the U. S. Office of Education, 1965. (b)

———. Final report: The development of the Bell System first aid and personal safety course. Palo Alto, Calif.: American Institutes for Research, 1967. (AIR-E81-4/67-FR)

Skinner, B. F. The science of learning and the art of teaching. *Harvard educ. Rev.*, 1964, 24, 86-97.